Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages

Cyndi Rader, Gina Cherry, Cathy Brand, Alexander Repenning and Clayton Lewis

University of Colorado

Campus Box 430, Boulder, CO 80309-0430

{crader, brand, gina, ralex, clayton}@cs.colorado.edu

Abstract

Designing programming languages that are accessible to elementary school children is a complex task. Programming languages that contain visual elements provide a good starting point, because they are inherently appealing to many children. As novice users, however, children require additional support to use programming languages effectively. In this paper we describe five principles for designing end-user programming languages which address some of the obstacles we have observed when users attempt to create sophisticated programs. The principles are based on extensions we have made to Visual AgentTalk, the tactile programming component of the Agentsheets system. Although our research has centered on children, we believe that the discussion is widely applicable to the creation of languages for novice users of any age.

Introduction

Over the past three years, we have worked with children using visual languages to create science models as part of the Science Theater/Teatro de Ciencias (sTc) project. For the first two years of this project, the children worked with KidSim/Cocoa, a pictorial language that uses programming by demonstration to create graphical rewrite rules [4]. In the third year, children used Visual AgentTalk (VAT), a partly visual, partly textual interface to the Agentsheets system [25].

Many children are attracted by the dynamic, visual aspects of these computing environments. They enjoy drawing their own representations of objects they use in their models, and they can describe in a general way how these objects should behave. However, they often face major challenges when trying to translate their ideas into workable computer programs [2]

Children have difficulty in both program comprehension and program creation. Program comprehension, the ability to determine the effects of a program command, requires users to understand: a) what each condition tests, b) what each action does, c) how conditions and actions are linked, and d) when and how rules are evaluated and executed (i.e., rule ordering). Program composition, the ability to assemble the language primitives in order to achieve the desired effects, presents even more challenges for the user. While children and other novice users can for the most part comprehend the if-then structure of rules, they have a difficult time mapping their own ideas into rules [11]. Users have trouble determining what conditions and actions to use, as well as which conditions to combine with which actions. They also dislike writing many rules to accomplish seemingly simple tasks, especially if the rules are repetitive.

Repenning and Ambach addressed some of these issues with their notion of tactile programming [23, 24]. Based on our experience working with children, we have extended their model of tactile programming to further address these issues, and have formulated a number of guidelines for designing end-user languages that contain a mixture of pictures and text. We will situate the discussion of these guidelines in the context of two specific units from a life sciences curriculum.

Setting

We have used models created in Agentsheets/VAT as part of a life sciences curriculum for a 4th/5th-grade class. Students in this class range in age between 8 and 11. They have used the software as part of regular classroom activities rather than in a computer club context. In this constructionist setting [20] our ultimate goal is to expand students' understanding of science concepts and creation of scientific explanations. The idea of using programming as learning vehicle is not new [6, 8, 13, 19, 26] but gets under increasing scrutiny because of high cost/benefit ratios. It is absolutely crucial to maintain a balance between learning about the software and learning about the science content. To achieve this, students progressed from exploring science models we created, to modifying the behavior of objects within our models, to adding new objects to an existing model, and finally to creating a model where they designed and coded most of the objects themselves. In this

way the children were introduced in stages to the relevant programming constructs while at the same time progressing in their science study.

A gradual introduction to programming in a specific learning context is often called *scaffolding*. In earlier educational programming languages such as Logo [3, 19] scaffolding was an approach to structure learning activities but was not part of the programming environment itself. Boxer moved beyond basic Logo by providing an integrated programming environment combining programming language with content material [5]. Others have scaffolded programming visually either by using iconic languages [30] or by combining graphical rewrite rules with flavors of programming by example [1, 10, 14, 17, 22, 27-29]. Guzdial added interactive scaffolding machinery to programming environments [12] to help users to understand programming related problem solving. The scaffolding principles presented in this paper are concerned with language design issues combining textual and iconic representations to increase the effectiveness of programming for end-users who are not interested in programming per se.

We will be using examples from two of the units in the curriculum to illustrate our language design guidelines. In the first unit, which we will refer to as the Energy Web (Figure 1), students added their own predators to a model of plants and animals in a food web which we created.

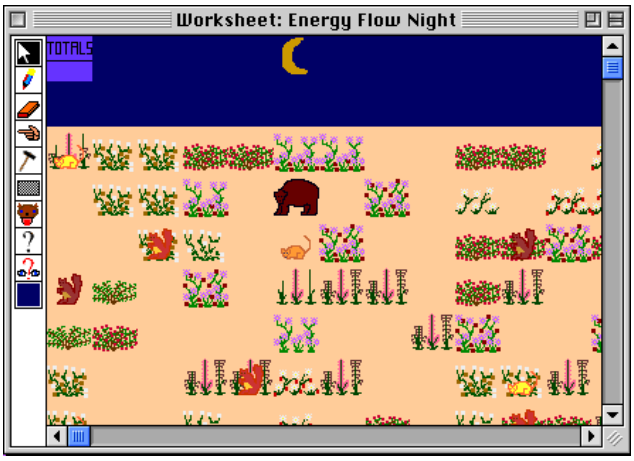


Figure 1. A FoodWeb Simulation world containing animal agents programmed by elementary school children.

In the second unit, which we will call Fly Catcher (Figure 10), students programmed their own carnivorous plants. We provided a model fly, and their task was to design a model plant that would be capable of “catching” the fly.

Design Principles

We present five language design guidelines that address the issues raised above.

1) Syntonicity. Programmers will be more successful if they can identify with the object they are programming.

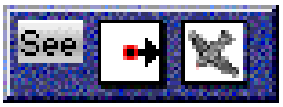
Papert [19] used the term *syntonicity* to explain why turtle geometry is easy for children to learn. He claimed that children could learn turtle geometry more easily because they could physically identify with the turtle. In contrast to anthropomorphism [15], in which computers are identified as human-like entities, syntonicity is about humans identifying themselves as part of a computational world such as an object on a computer screen or the program controlling it.

More recently, Watt [31] has expanded on the importance of syntonicity in programming. He asserts that people think about programs in psychological, rather than purely computational, terms. Watt thinks of syntonicity not only in physical terms, but also in terms of goals, wants, and intentions. Syntonicity may be particularly important for children, who can understand that other beings have intentional states, but do not yet have the capacity for the sophisticated abstract reasoning needed to think computationally.

Visual AgenTalk's tactile programming paradigm aids syntonicity. People can identify with VAT programs because they can assume the role of the program executing device through the manipulation of tactile language objects [23]. This ability allows users to playfully explore and to comprehend language functionality. In contrast to languages consisting of graphical rewrite rules, the mixture of text and icons allows the user to view the rules from the point of view of a specific agent.

Our work with Visual AgentTalk supports the idea that syntonicity is an important consideration for designing languages. We found that even small, seemingly trivial changes to language primitives can result in considerable improvement of program comprehension. For example, early in the language design process, we noticed that children had a difficult time understanding conditions in which the actor was not stated explicitly. An example of this idea is the original "see" condition (Figure 2), which checks to see if there is an agent with a given appearance in a specified direction. We changed this condition to "I-see" because children had an easier time understanding the condition when it better supported taking on the perspective of the agent doing the seeing.

a)



b)

Figure 2: Syntonicity helps users to identify with program. a) Original language condition to detect other agents, b) syntonic variant of condition.

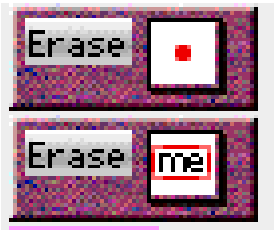


Figure 3: "me" also adds to syntonicity in Erase command

The erase command (Figure 3) is another feature that we changed in order to support physical syntonicity. The original erase command required the programmer to choose a direction from a pop-up menu of arrows which denote positions in the grid relative to the agent enacting the command. The command to erase oneself looked like "erase ." where" ." meant "the agent in my square", i.e., the agent enacting the command. Children found this notation confusing, so we changed the command to be more explicitly self-referential by replacing the "." icon with a "me" icon.

We also found that representations which allow children to identify with the internal state of an agent make program comprehension and composition easier. For example, in the energy web simulation, children needed a condition to determine whether an animal was hungry. In the original language design (Figure 4), this would be accomplished by testing that the value of a Boolean variable is equal to 1. This syntax did not support thinking of hunger as an internal state. In order to make this condition more comprehensible to children, we created two conditions to deal with Boolean or state variables. The new conditions - "I am hungry" and "I am not hungry" are a more natural way to express hunger as an internal state of the agent.

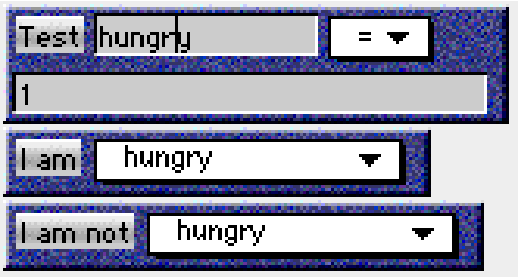


Figure 4: Syntonicity includes goals, wants and intentions

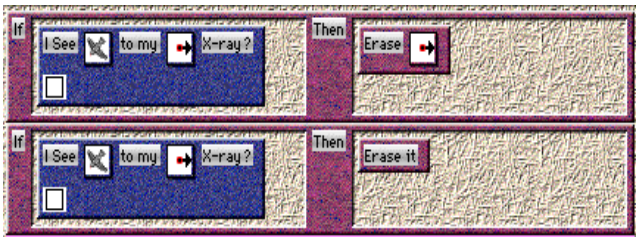


Figure 5: The pronoun "it" can link conditions and actions.

2) Phrase Structure. A mixed textual and iconic representation, if structured as phrase, can significantly increase readability of programs over purely iconic representations.

While syntonicity helps users to identify with the objects being programmed, complete phrases enable users to more easily state their intentions. Users find it easier to understand and create rules that are written as complete, grammatical sentences. Sometimes just changing the order of the parameters and adding extra text to the command format can make a big difference in comprehension. The See condition mentioned above (Figure 5) provides an example. Notice that, in addition to adding the actor (I), the order of the parameters is changed and two words are added. The condition as it now reads matches more closely how we speak. Thus, "See right (a bird)" becomes "I see (a bird) to my right."

The combination of adding the actor "I", and making conditions self-referential using "me", resulted in the unnatural condition "I See me (depiction)" for checking one's own appearance. Since this command was confusing to the children, we created a separate command for checking one's own appearance, called "I look like" (Figure 6)

Figure 6: Conditions that match how users talk are easier to understand.

Users were also confused by actions in which the object was implied rather than stated explicitly. For example, figure 5 shows a rule that erases an object to an agent's right. The object of the action (i.e., the agent being erased) is a bird. The user must mentally connect the object "seen" by the condition with the object being erased, a connection which many users find difficult to make. This issue can be resolved by using the pronoun "it" to connect the condition and action of a rule. The rule can then be stated more naturally as "If I see a bird to my right then erase it." This format also eliminates one common novice programming error - neglecting to make the position arrows match. For example, students might mistakenly enter a rule that said

"If I see a bird to my right then erase (the agent) to my left."

Designing a language at the level of phrases, rather than just individual commands, moves the designer closer to the realm of the end user.

3) Sets. Allowing users to operate on sets of agents can simplify problem solutions and reduce rule explosion.

Pane [18] has found that children very rarely describe a solution that uses loops when asked to devise solutions to a problem,. Instead, children seem more comfortable expressing their ideas as operations on sets of objects. We have found that sets can be very useful in helping users create more powerful programs.

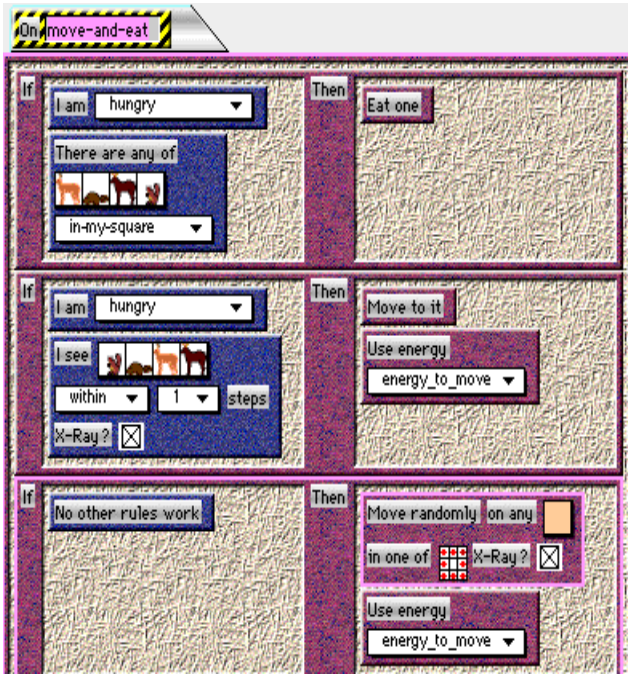


Figure 7: A student uses sets to select prey for her mountain lion.

Children, as well as most adults, have little patience for creating simulations in which it takes many similar rules to accomplish what is to them conceptually one task. For example, in the Energy Web, students created predators that could potentially eat many different types of food. Without sets, users would have to specify each type of food in a separate rule; however, using sets of depictions (agent appearances), students could easily specify the eating behavior in a single rule. In addition to eating several different types of foods, animals also look or "hunt" for different types of foods (see Figure 7). Again, without sets, each type of food would require a separate rule. Creating a potentially large number of rules which are all conceptually related is a tedious task. Program readability is also reduced, because of the need to scroll through a large number of rules and to mentally connect those rules which represent a single concept.

Sets can also help address some issues regarding rule order and random behavior. In the example above, if the eat rule were written as a series of separate rules, the predator would prefer to eat some foods over others, depending on the order in which the rules were written. Expressing eating behavior as a set provides a simple way to express equal preference for the possible types of food because one element from the set is randomly chosen to eat.

Sets of directions are a natural way to express random movement. For example, the third rule displayed in Figure 7 allows the user to "Move randomly on" a set of agents (in this case, ground), in any of a set of directions. For this particular example, it makes most sense to move in any of the eight possible directions (the default). In some cases, the user might select a subset of the eight directions. For example, the fly in the Fly Catcher program has both a right-facing and left-facing appearance. Figure 8 shows two rules that specify how the fly should move, based on its appearance.

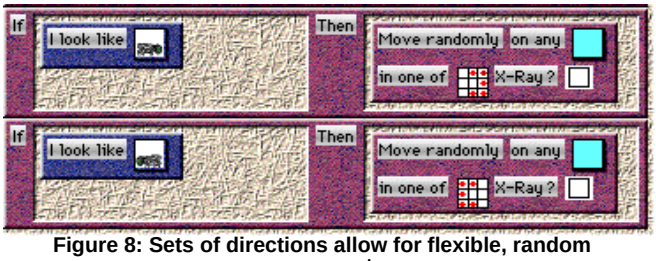


Figure 8: Sets of directions allow for flexible, random movement.

The corresponding solution in Cocoa or Agentsheets graphical rewrite rules would have required a much larger number of rules and despite the similarity of the rules each of these rules would have required a complete programming by demonstration cycle.

4) Domain Orientation. Language pieces tailored to the modeling domain help users to bridge the problem-solution gap by hinting at solutions.

Fischer [7, 9] has argued that domain orientation is beneficial for end users because it reduces the conceptual distance between the problem-domain semantics and the software artifact. A program stated in the language of the problem domain will generally be more understandable to users because it matches the way they naturally talk about the domain. Programming language scaffolding that is domain-oriented [25] can decrease the amount of time required for users to create a program and also potentially increase the complexity of what the users are able to create. We have employed domain-orientation in three areas

within the VAT environment: commands, variables, and templates.

4.1 Domain-Oriented Commands

We have reported previously that children often have difficulty mapping the science content they wish to express onto the operations provided by a visual language [2]. We have been working toward reducing the distance between the way children describe their model and the commands available to implement the model. We hoped that reducing this distance would help children construct models that were more complex and more explanatory than those done in previous years using a completely pictorial language. As recommended by Lewis & Olson [16], we have moved away from the traditional computer science perspective that a handful of simple general primitives should be combined in creative ways by the programmer to achieve a desired effect.

Instead, we have supplied commands which more closely match the way children describe their agents' actions. This difference can be as simple as replacing, for example, "erase the squirrel to my right" with "eat the squirrel to my right" and" erase me" with "die." Although "eat" and "die" map to the same lisp code as erase, for a child without much programming experience, "eat" and "die" are more obvious choices than "erase."

We have also provided some special-purpose commands to perform actions which are common for the specific problem domain and which could be tedious (and not intuitive) for users to compose using lower-level primitives. For instance, a student may write a rule that says "If I see (a rock) within (1-4) steps, then hide underneath it". The "hide underneath it" action moves the agent to the grid square containing the rock (or specified object), sets a variable in the agent which marks it as hidden and actually places it underneath the rock (i.e., beneath the rock in the stack of objects on that grid square). "Hide underneath it" matches the way the students describe an animal's behavior so it's an easy choice when they want to implement this behavior. This command also hides the sometimes confusing process of setting a variable.

4.2 Domain-Oriented Variables

Using variables can greatly expand what can be accomplished in a simulation. For example, a common requirement is that agents change over time: plants grow, baby animals mature, etc. In our experience, children can manipulate variables, but they have difficulty determining when and how to use them within their own programs [21]. By providing a pull-down menu from which children can choose domain-oriented variables, we can increase the ability of children to understand and use variables. For example, in the Energy Web unit, animals had an "energy" value that was increased by eating and decreased by other activities, such as moving (see Figure 11) Animals would die if they were "out of" energy (i.e., energy <= 0).

In addition to numeric variables, some simulations may rely on Boolean or state variables. In the Fly Catcher project, students could make their plant parts be sticky or slippery. Figure 9 shows one solution in which the plant part is made to be both "slippery" and colored red (to attract the fly). The plant designed by this student is shown in figure 10.

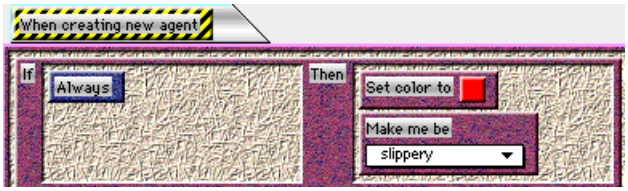


Figure 9: Domain-Oriented variables are easier to use.

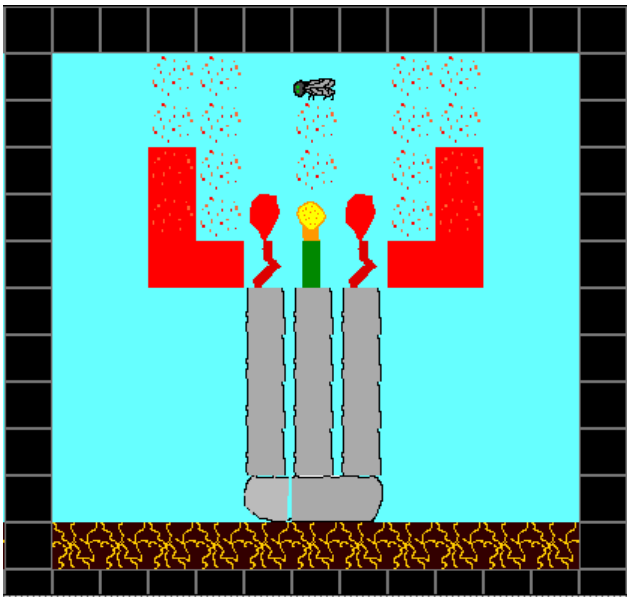


Figure 10: Student's fly catcher model.

4.3 Domain-Oriented Templates

Two particularly difficult issues for students are combining conditions and actions correctly and understanding rule order. Although framing rules as complete, grammatical sentences helps, selecting the necessary primitives can still be a difficult and error-prone process. In VAT, we are able to provide a "template" agent that the students can "clone." A template is an agent with predefined methods that relate to the problem domain. The methods will typically contain domain-oriented commands, which may rely on domain-oriented variables. Unlike commands in regular agents, however, the commands in the templates often have information that is left blank. When students clone a template, they create a new agent which they customize by assigning a name, creating one or more depictions, and entering the necessary information into the agent's commands.

Templates are used in the Energy Web to allow the students to focus on the science content of their models (i.e., the specification of the animal, particularly what it eats) and to ignore the extra methods which were required for overall program operation but were not important in terms of the science content. Figure 11 shows a prepared Energy Web template. This template contains the same three rules as the student's rules in figure 7, except the choice of animals is left blank. Additional methods (not shown) perform tasks such as setting initial parameters, checking energy levels and updating displays which show how much energy an animal has remaining.

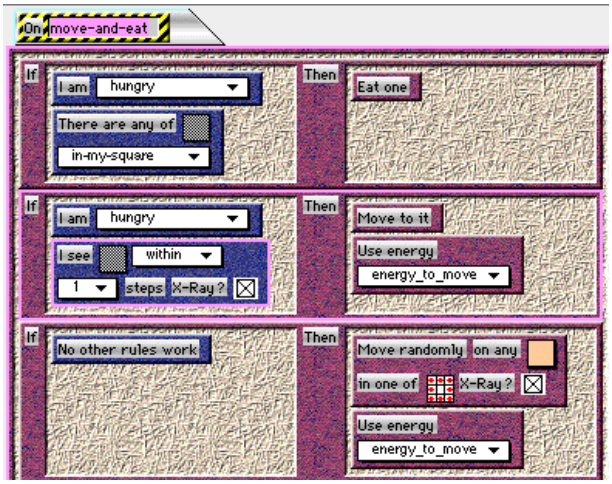


Figure 11: Templates allow students to focus on important science content, while avoiding problems with command selection and rule ordering.

By placing the blank rules in a template, we eliminate the need for the students to select the necessary conditions and actions. We also avoid a common rule-ordering problem. In rule-based systems like VAT, rules are tested from the top down, and the first rule with a "true" condition is executed. The first two rules in the template represent "eating" and "hunting." Since animals must eat in order to survive, they should have a preference for eating (i.e., the "eat" rule should be first). However, students tend to think of the actions as a sequence. Therefore, when working without a template, they often place the "hunt" rule before the "eat " rule.

Since the templates are constructed of ordinary VAT methods with some rules preselected, students are not constrained to just the behaviors provided by the template. They can add and delete rules in the same manner as if they created the agent from scratch. For example, the student whose model is shown in figure 11 extended the template so that her animal would "sleep" at night.

5) Selection and Polymorphism. Composition is supported by offering users a large selection of powerful commands and clustering them appropriately.

We frequently rephrased a single command to apply it in slightly different situations. For example, an action may apply to me, to all my neighbors, to one of or any of a set of agents. The same action may apply to an agent in my grid-square or in a certain direction or a certain distance away. The action may apply to the agent only if it appears on top of the stack of agents in a grid-square or if it appears anywhere in the stack. We could have designed one super command with multiple options which could be used in all of the above situations. However, we have found that the simpler and more grammatical the phrasing is of the command, the more readily children understand it. Providing five or six variant versions of the same basic command helps them better comprehend each individual command. It also prevents them from stumbling over how to set complicated options. Figure 12 shows a palette with four ways to use the "tell" command.

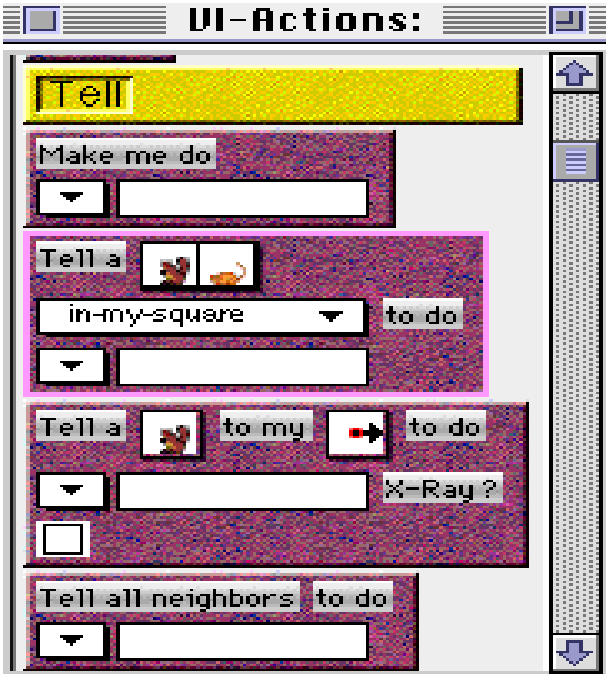


Figure 12: Polymorphic commands to tell an agent to perform a method. The drop-down arrow indicates that the user may choose the method from a list or type in a method name.

The creation of multiple domain-oriented commands with different names but similar functionality, and the creation of multiple versions of polymorphic commands rapidly leads to impossibly large command palettes. We have reduced the choices for students by making available only those commands which might be useful for the current modeling project. We have further subdivided the available commands by grouping them in separate palettes. Clicking on a rule background, the normal method for

bringing up a palette, makes the simplest and most commonly used commands appear. By clicking on the Tools menu, students can access the more advanced and developer command palettes. Within each palette, commands are organized into categories separated by headers labeled with a domain-oriented name.

We have found that children have varying tolerances for lists of commands. Some children demand almost continual adult guidance when looking for a command and seem to have difficulty reading all the way through a list. Therefore, we have kept the basic palettes rather short. The most enthusiastic programmers among the students, however, preferred to use the developer's palette because they liked having a large number of options.

Conclusion

In a sense, the principles we articulate here represent an evolution from simplicity to complexity: we have replaced a simple and spare language with a larger and more complex one with some quite specialized features. But this elaboration has moved the language closer to children's conceptions of their problems. We have found that when we provided scaffolding in accordance with the guidelines suggested in this paper, children were more successful creating models. However, there is still room for improvement. About half of the children were able to build model fly catchers with little adult help, which represents progress over our earlier efforts. These are good results for a group of children in a real classroom, with a typical range of backgrounds, not a self-selected sample. About half of the children needed a good deal of one-on-one attention to succeed. Broadly speaking, the issues that continue to cause problems are related to those we have addressed: children have trouble when their intuitive representations of behavior need to be dissected and recast to fit the structure of the language. Despite a number of remaining challenges end-user programming with proper scaffolding can be an effective and engaging teaching vehicle. We observed that over half of the class returned on a volunteer basis back to their simulations during lunch breaks.

Acknowledgments

The authors would like to thank the entire sTc team: Heidi Carlone, Linda Hagen, Teresa Garcia, Cory Buxton, Carlos Garcia, Page Pulver, Mike Eisenberg, Mary Lou Salazar, and Steve Guberman. Our work is funded under the National Science Foundation Advanced Applications of Technology program, the Advanced Research Projects Agency and the Technical Reinvestment Program.

References

1. Bell, B., and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," *1993 IEEE Workshop on Visual Languages*, Bergen, Norway, IEEE Computer Society Press, 1993, pp. 188-195.

2. Brand, C., and C. Rader, "How Does a Visual Simulation Program Support Students Creating Science Models?," *Proceedings of the 1996 IEEE Symposium of Visual Languages*, Boulder, CO, Computer Society, 1996, pp. 102-109.

3. Cohen, R. S., "Logo in the Primary Classroom: Should Simplified Versions Be Used?," *The Computer Teacher*, pp. 41-43, 1990.

4. Cypher, A., and D. C. Smith, "KidSim: End User Programming of Simulations," *Proceedings of the 1995 Conference of Human Factors in Computing Systems*, Denver, CO, ACM Press, 1995, pp. 351-358.

5. diSessa, A. A., "A Principled Design for an Integrated Computational Environment," *Human-Computer Interaction*, Vol. 1, pp. 1-47, 1985.

6. diSessa, A. A., "An Overview of Boxer," *Journal of Mathematical Behavior*, pp. 3-15, 1991.

7. Eden, H., M. Eisenberg, G. Fischer, and A. Repenning, "Domain-Oriented Design Environments: Making Learning a Part of Life," *Communications of the ACM*, Vol. 39, pp. 40-42, 1996.

8. Fenton, J., and K. Beck, "Playground: An Object-Oriented Simulation System with Agent Rules for Children of All Ages," *OOPSLA 89*, New Orleans, LA, ACM Press, 1989, pp. 123-137.

9. Fischer, G., "Domain-Oriented Design Environments," in *Automated Software Engineering*, Ed., Kluwer Academic Publishers, Boston, MA, 1994, pp. 177-203.

10. Furnas, G. W., "New Graphical Reasoning Models for Understanding Graphical Interfaces," *Proceedings CHI'91*, New Orleans, LA, ACM Press, 1991, pp. 71-78.

11. Gilmore, D., K. Pheasey, J. Underwood, and G. Underwood, "Learning graphical programming: An evaluation of KidSim," *Proceedings of the Fifth IFIP Conference on Human-Computer Interaction*, London, 1995, pp. .

12. Guzdial, M., "Software-Realized Scaffolding to Facilitate Programming for Science Learning," *Interactive Learning Environments*, pp. 1994.

13. Kahn, K., "Seeing Systolic Computations in a Video Game World," *Proceedings of the 1996 IEEE Symposium of Visual Languages*, Boulder, CO, Computer Society, 1996, pp. 95-101.

14. Kirsch, R., A., "Computer Interpretation of English and Text and Picture Patterns," *IEEE Transactions on Electronic Computers*, Vol. 13, pp. 363-376, 1964.

15. Laurel, B., *Computers as Theater*, Addison-Wesley Publishing Company, Reading, MA, 1993.

16. Lewis, C., and G. M. Olson, "Can Principles of Cognition Lower the Barriers to Programming?," *Empirical Studies of Programmers: Second Workshop*, Norwood, NJ, Ablex Publishing, 1987, pp. 248-263.

17. Lieberman, H., "An Example-Based Environment for Beginning Programmers," in *Artificial Intelligence and Education*, R. W. Lawler and M. Yazdani, Ed., Ablex Publishing, Norwood, NJ, 1987, pp. 135-151.

18. Pane, J., "A Programming System for Children that is Designed for Usability," *Presented at 7th Workshop on Empirical Studies of Programmers: Graduate Student Workshop*, Alexandria, VA, 1997.

19. Papert, S., *Mindstorms: Children, Computers and Powerful Ideas*, Basic Books, New York, 1980.

20. Papert, S., and I. Harel, Eds., *Constructionism*, Ablex Publishing Corporation, Norwood, NJ, 1993.

21. Rader, C., C. Brand, and C. Lewis, "Degrees of Comprehension: Children's Understanding of a Visual Programming Environment," *Proceedings of the 1997 Conference of Human Factors in Computing Systems*, Atlanta, GA, ACM Press, 1997, pp. 351-358.

22. Repenning, A., "Bending the Rules: Steps toward Semantically Enriched Graphical Rewrite Rules," *Proceedings of Visual Languages*, Darmstadt, Germany, IEEE Computer Society, 1995, pp. 226-233.

23. Repenning, A., and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," *Proceedings of the 1996 IEEE Symposium of Visual Languages*, Boulder, CO, Computer Society, 1996, pp. 102-109.

24. Repenning, A., and A. Ioannidou, "Behavior Processors: Layers between End-Users and Java Virtual Machines," *Proceedings of the 1997 IEEE Symposium of Visual Languages*, Capri, Italy, Computer Society, 1997, pp. 402-409.

25. Repenning, A., and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer*, Vol. 28, pp. 17-25, 1995.

26. Resnik, M., "Beyond the Centralized Mindset: Explorations in Massively-Parallel Microworld," Massachusetts Institute of Technology, Ph.D. dissertation, Dept. of Computer Science, Pages, 1992.

27. Robinson, R., D. Cook, and S. Tanimoto, "Programming Agents with Visual Rules," *Proceeding of Visual Languages*, Darmstadt, Germany, IEEE Computer Scociety, 1995, pp. 13-20.

28. Siromoney, G., R. Siromoney, and K. Krithivasan, "Picture Languages with Array Rewriting Rules," *Information and Control*, pp. 447-470, 1973.

29. Smith, D. C., A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM*, Vol. 37, pp. 54-68, 1994.

30. Tantimoto, S., and M. Runyan, "PLAY: An Iconic Programming System for Children," *Visual Languages*, pp. 191-205, 1986.

31. Watt, S., "Syntonicity and the Pshychology of Programming," *Proceedings of the Tenth Annual Meeting of the Pshychology of Programming Interest Group*, Milton Keenes, UK, Knowledge Media Institute, 1998, pp. 75-86.